COMP 251 - Greedy Algorithms

Algorithms and Data Structures

Written By

David Knapik

McGill University
david.knapik@mcgill.ca

February 26, 2018

Contents

1

Introduction - Scheduling

Greedy algorithms make locally optimal (**myopic**) choices at each step. They are fast, simple, and easy to code BUT usually are trash.

1.1 Task Scheduling

Suppose a firm receives job orders from n customers. This firm can only do one task at a time. Let the time it takes to complete the job of customer i be denoted t_i. Each customer wants their job done as fast as possible. Assume the firm processes the jobs in the order $\{1, 2, ..., n\}$ and that the wait time of customer l is

$$w_l = \sum_{i=1}^{l} t_i$$

To be capitalist and maximize the profits, the objective of the firm is to minimize the sum of the waiting times

$$\sum_{l=1}^{n} w_l = \sum_{l=1}^{n} \sum_{i=1}^{l} t_i$$

Algorithm 1.1.1. Greedy Task Scheduling Algorithm:

1. Sort the jobs by length $t_1 \leq t_2 \leq \cdots \leq t_n$.
2. Schedule jobs in the order $\{1, 2, ..., n\}$.

Theorem 1.1.1. *The above Greedy Task Scheduling Algorithm works and is optimal.*

Proof. A commonly used way to prove that Greedy Algorithms work is by using an exchange argument. We shall do so here.

Let the greedy algorithm schedule in the order $\{1, 2, ..., n\}$. Assume $\exists$ a better schedule $\mathcal{S}$. Then, there is a pair of jobs i, j such that

1. Job i is scheduled immediately before job j by $\mathcal{S}$
2. Job i is longer than job j: $t_i > t_j$

But then, exchanging the order of jobs i, j gives a better schedule $\hat{\mathcal{S}}$. Observe that the waiting time of the other jobs remains the same i.e. $\hat{w_k} = w_k \ \forall k \neq i, j$. But, clearly $\hat{w_i} + \hat{w_j} < w_i + w_j$. This contradicts that $\mathcal{S}$ was optimal. □

1.1.1 Run-time

We just needed to sort n time lengths. Thus, the run-time is $O(n \log n)$. So, the Greedy Task Scheduling algorithm is efficient.

1.2 Class Scheduling (an Interval Selection Problem)

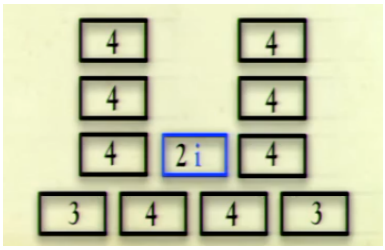
Suppose there is a set $I = \{1, ..., n\}$ of classes that request to use a room. Class i has start time s_i and finish time f_i. We want to book as many classes into the room as possible. However, we cannot book 2 classes that need to use the room at exactly the same time.

What kinda of greedy algorithm would we use for this? Here are some options:

1. First-Start: we select the class that starts earliest and iterate on the remaining classes that do not conflict with this selection. This algorithm is bad and doesn't work. For example, the first class could be all day long

2. Shortest duration: we select the class of shortest duration, and iterate on the remaining classes that do not conflict with this selection. This algorithm doesn't work. For example we could have a bad ordering and thus many clashes.

3. Minimum-Conflict: we select the class that conflicts with the fewest number of classes and then iterate. This algorithm doesn't work.



4. Last-Start: we select the class that starts last and iterate.

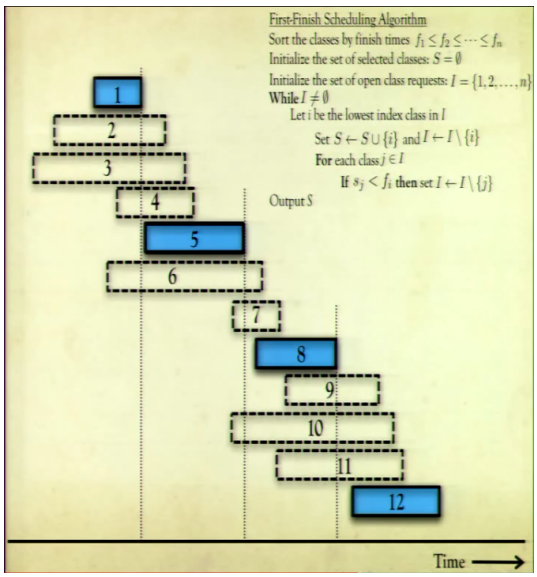
Last-Start actually works and outputs an optimal schedule. It is symmetric to First-Finish where we select the class that finishes first and iterate on the classes that do not conflict with this selection.

Algorithm 1.2.1. FirstFinish(I)

1. Let Class 1 be the class with the earliest finish time.

2. Let X be the set of classes that clash with Class 1.

3. Output $\{1\} \cup$ FirstFinish $(I \setminus X)$.



Theorem 1.2.1. *The FirstFinish scheduling algorithm outputs an optimal schedule.*

Before we prove this, we present the following lemma:

Lemma 1.2.1. There is an optimal schedule that selects Class 1.

Proof. Take an optimal schedule $\mathcal{S}$ with $1 \notin \mathcal{S}$. Let i denote the lowest index class selected in $\mathcal{S}$.

We claim that $(\mathcal{S} \setminus \{i\}) \cup \{i\}$ is a feasible allocation of max. size.

Indeed this follows since: $f_i \leq s_j$ for any $j \in \mathcal{S} \setminus \{i\}$. But $f_1 \leq f_i \leq s_j$ so Class 1 doesn't conflict with any class in $\mathcal{S} \setminus \{i\}$. Thus, $(\mathcal{S} \setminus \{i\}) \cup \{i\}$ is feasible with cardinality equal to $|\mathcal{S}|$. □

Now, back to the proof of the theorem:

Proof. We proceed via induction on $|opt(I)|$. Our base case is that $|opt(I)| = 1$ and clearly FirstFinish outputs $\{1\}$ so we are done. Now our hypothesis is that if $|opt(I)| = k$ then FirstFinish gives an optimal solution. Let $|opt(I)| = k + 1$.

FirstFinish outputs $\{1\} \cup$ FirstFinish$(I \setminus X)$. By our lemma, $\{1\} \in \mathcal{S}^*$ for some optimal solution $\mathcal{S}^*$. Thus $\mathcal{S}^* \setminus \{1\}$ is an optimal solution for the sub-problem $I \setminus X$. So $|opt(I \setminus X)| = k$. And by the induction hypothesis, FirstFinish gives an optimal solution for the sub-problem $I \setminus X$. $\rightarrow |FirstFinish(I \setminus X)| = k$ and thus

$$|\{1\} \cup FirstFinish(I \setminus X)| = k + 1$$

□

1.2.1 Runtime

There are at most n iterations and it takes $O(n)$ time to find the class that finishes earliest in each iteration. Thus, the run-time is $O(n^2)$. But, this was a very crude analysis. With a more subtle implementation and analysis, we can actually get $O(n \log n)$ (exercise!).

2

The Shortest Path Problem

Say that we have a directed graph $G = (V, A)$ with a source vertex $s \in V$. Let each arc $a \in A$ have non-negative length l_a. The length of a path P in the graph is then $l(P) = \sum_{a \in P} l_a$. The problem we consider here is that we want to find the shortest length paths from s to every other vertex $v \in V$. Amazingly, we can actually do this in 1 go, rather than running an algorithm say n times. We do this via the greedy algorithm know as Dijkstra's Shortest Path Algorithm which was conceived by the Dutch computer scientist Edsger W. Dijkstra in 1956.

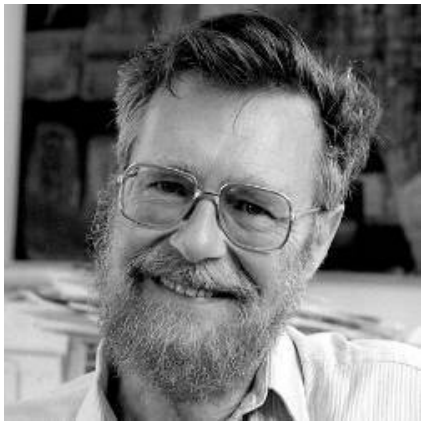


Figure 2.1: Edsger W. Dijkstra

2.1 Dijkstra's Shortest Path Algorithm

Algorithm 2.1.1.

Set $\mathcal{S} = \emptyset$
Set $d(s) = 0$ and $d(v) = \infty, \quad \forall v \in V \setminus \{s\}$
While $\mathcal{S} \neq V$
  If $v = \underset{u \in V \setminus \mathcal{S}}{\operatorname{argmin}} \, d(u)$ then set $\mathcal{S} \leftarrow \mathcal{S} \cup \{v\}$
  For each arc (v, w):
    If $d(w) > d(v) + \ell_{vw}$ and $w \in V \setminus \mathcal{S}$
      then set $d(w) = d(v) + \ell_{vw}$

This algorithm only finds the shortest path distances. But, we can easily enhance the algorithm to keep track of the shortest path themselves as well:

Algorithm 2.1.2.

Set $\mathcal{S} = \emptyset$
Set $\mathcal{T} = \emptyset$, $\text{pred}(s) \leftarrow$ "$-$", and $\text{pred}(v) \leftarrow \square$
Set $d(s) = 0$ and $d(v) = \infty, \quad \forall v \in V \setminus \{s\}$
While $\mathcal{S} \neq V$
  If $v = \underset{u \in V \setminus \mathcal{S}}{\operatorname{argmin}} \, d(u)$
    then set $\mathcal{S} \leftarrow \mathcal{S} \cup \{v\}$ and $\mathcal{T} \leftarrow \mathcal{T} \cup (\text{pred}(v), v)$
  For each arc (v, w):
    If $d(w) > d(v) + \ell_{vw}$ and $w \in V \setminus \mathcal{S}$
      then set $d(w) = d(v) + \ell_{vw}$
      $\text{pred}(w) \leftarrow v$

See the example in the lecture and make sure you understand how this algorithm works!

2.1.1 Special Case reduced to BFS

If each arclength is equal to 1, we simply get BFS. Try it!

2.1.2 Does it Work?

First observe that $\mathcal{S}$ is a set of vertices and $\mathcal{T}$ is a set of arcs. Thus, we introduce the following notation:

1. Let $\mathcal{S}^k$ denote the set of vertices in $\mathcal{S}$ at the end of the k-th iteration.
2. Let $\mathcal{T}^k$ denote the set of arcs in $\mathcal{T}$ at the end of the k-th iteration

Observe that the arcs in $\mathcal{T}^k$ are all between vertices in $\mathcal{S}^k$. Thus, $\mathcal{G}^k = (\mathcal{S}^k, \mathcal{T}^k)$ is actually a directed graph.

Remark. Note that $\mathcal{G} = \mathcal{G}^n$ is the final output of Dijkstra's algorithm.

In fact, it turns out that $\mathcal{G}$ is an **arborescence** (a directed tree) rooted at the source node s:

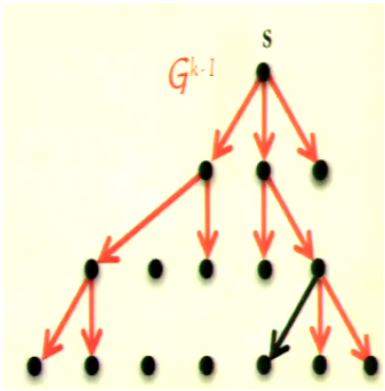
Theorem 2.1.1. *$\mathcal{G}^k$ is an arborescence rooted at s*

Proof. Base case is $k = 1$. Well, $\mathcal{S}^1$ only contains s. But $\mathcal{T}^1$ contains no arcs since $pred(s) =$ "$-$". Thus, $\mathcal{G}^1$ is trivially an arborescence rooted at s.

Now, our hypothesis is that $\mathcal{G}^{k-1}$ is an arborescence rooted at s. Consider $\mathcal{G}^k$. Let v_k be the vertex added to $\mathcal{S}$ in the k-th iteration. Thus:

$$\mathcal{S}^k = \mathcal{S}^{k-1} \cup \{v_k\} \text{ and } \mathcal{T}^k = \mathcal{T}^{k-1} \cup (pred^{k-1}(v_k), v_k)$$

So, v_k has in degree $=1$ in $\mathcal{G}^k$ and outdegree $= 0$. Furthermore, by definition, $pred^{k-1}(v_k) \in \mathcal{S}^{k-1}$. Thus, $\mathcal{G}^k$ is an arborescnece rooted at s with v_k as a leaf.



□

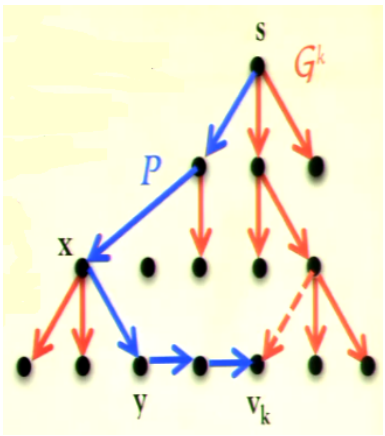
So, at this point, we know that Dijkstra's algorithm outputs an arborescence. But moreover, we claim that the paths induced by the arborescence are shortest paths:

Theorem 2.1.2. *The graph $\mathcal{G}^k$ gives the true shortest path distances from s to every vertex in $\mathcal{S}^k$*

Proof. The base case $k = 1$ is trivial since $d^1(s) = 0 = d^*(s)$. (Here we use the notation that d^* denotes the true shortest path distances.)

Now, for our hypothesis assume $\mathcal{G}^{k-1}$ gives the shortest distances from s to every vertex in $\mathcal{S}^{k-1}$ i.e. $d^{k-1}(v) = d^*(v)$ for all $v \in \mathcal{S}^{k-1}$.

Now, consider $\mathcal{G}^k$. Let v_k be the vertex added to $\mathcal{S}$ in the k-th iteration. We need to show that the new path is the shortest path to v_k. We argue via contradiction. Take the shortest path P from s to v_k with as many arcs in common with $\mathcal{G}^k$ as possible. Let x be the last vertex of $\mathcal{G}^{k-1}$ in P. Let $y \notin \mathcal{S}^k$ be the vertex after x in P. (Such a y $\exists$, or P is in $\mathcal{G}^k$ and we are done).



Since y is on the shortest path from s to v_k and the arclengths are nonnegative, we have:

$$d^*(y) \leq d^*(v_k) < d^k(v_k)$$

noting that we have strict inequality or else we are done. But, since $x \in \mathcal{S}^{k-1}$ we have:

$$\begin{aligned} d^k(y) &\leq d^{k-1}(x) + l(x,y) \\ &= d^*(x) + l(x,y) \quad \text{by the induction hypothesis} \\ &= d^*(y) \end{aligned}$$

Therefore, $d^k(y) \leq d^*(y) < d^k(v_k)$ which contradicts the selection of v_k as $d^k(y) < d^k(v_k)$. □

2.1.3 Runtime

We have n iterations and at most n distance updates at each iteration. Thus, the runtime is $O(n^2)$. Later in this course we will see how to implement Dijkstra's algorithm in $O(m \log n)$ using a heap (here m is the number of arcs).

3 Huffman Codes

Suppose we want to encode the alphabet in binary. How many bits do we need to be able to encode every letter? Well, 5 bits would suffice since $2^5 \geq 26$. But, is this good encoding? How do we measure the quality of this encoding? The most natural measure would be the length of the encoding. So if f_i is the frequency at which letter i appears in alphabet $\mathcal{A}$ then:

$$Cost = \sum_{i \in \mathcal{A}} l_i \cdot f_i$$

But, some letters appear more often than others in English, such as a and i. So, we could reduce the cost if more frequent letters have smaller representations than less frequent letters. Indeed, a well known code does exactly this:

3.1 Morse Code

By allowing a different number of bits per letter, only a max of 4 bits are required since $2^1+2^2+2^3+2^4 \geq 26$. However, we may think that there is a problem with Morse Code: its ambiguous. For example, $1101 = q$ but also $1101 = ma$. The way of fixing this is by using a "pause" or " $*$ " after each letter. This means that Morse Code is actually a ternary (not binary) code.

We are only interested in a binary code. So, is it possible to avoid ambiguities with a true binary code?

3.2 Prefix Codes

Definition 3.2.1. We say that a coding system is **prefix-free** if no codeword is a prefix of another codeword.

Remark. Morse code is NOT prefix free.

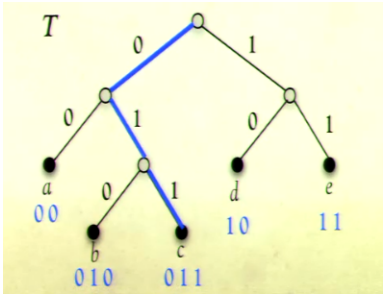
3.2.1 Binary Tree Representations

We can actually use a binary tree to represent a prefix-free binary code:

1. We have a binary tree T.
2. Each left edge has label 0 and each right edge has label 1.
3. The leaf vertices are the letter of the alphabet.

4. The codeword for a letter is the labels on the path from the root to the leaf.

Example 3.2.1.



Theorem 3.2.1. *A binary coding system is prefix-free iff it has a binary tree representation.*

Proof. ■ $\leftarrow$: The letters are the leaves, thus a path P_x from the root to leaf x and a path P_y to a leaf y must diverge at some point. Thus, the codeword for x cannot be a prefix of the codeword for y.

■ $\rightarrow$: Given a binary coding system, we define the binary tree recursively. A letter whose code word starts with a 0 is placed in the left subtree, otherwise it is placed in the right subtree. □

Our first important observation is the following:

Theorem 3.2.2 (Observation 1)**.**

$$Cost(T) = \sum_{i \in \mathcal{A}} f_i \cdot d_i(T)$$

where $d_i(T)$ denotes the depth of the leaf in the tree.

This first observation tells us that if we must use a tree of specified shape then it is easy to determine the best way to assign the letter to the leaves. Specifically, sort the letter in increasing order of frequency then iteratively assign the least frequent letter to the deepest leaf.

What Observation 1 doesn't tell us though, is what the best shape for the tree is. So, let

$$n_e = \sum_{i \in \mathcal{A}: e \in P_i} f_i$$

be the number of letters (weighted by frequency) whose root-leaf paths use edge e in T. Then, we have the following:

Theorem 3.2.3 (Observation 2)**.**

$$Cost(T) = \sum_{e \in T} n_e$$

Proof.

$$\begin{aligned} Cost(T) &= \sum_{i \in \mathcal{A}} f_i \cdot d_i(T) \\ &= \sum_{i \in \mathcal{A}} f_i \sum_{e: e \in P_i} 1 \\ &= \sum_{e \in T} \sum_{i \in \mathcal{A}: e \in P_i} f_i \\ &= \sum_{e \in T} n_e \end{aligned}$$

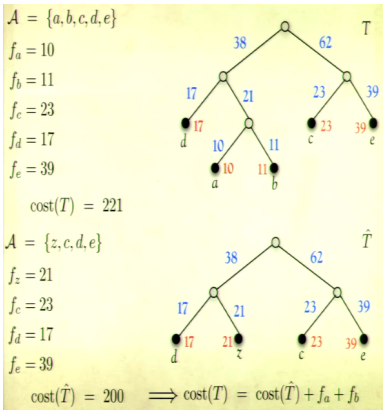
□

Also, we have the key formula:

Theorem 3.2.4 (Key Formula)**.** *Let $\hat{T}$ be the tree formed from T by removing a pair of sibling-leaves a and b and labelling their parent z where $f_z = f_a + f_b$. Then*

$$Cost(T) = Cost(\hat{T}) + f_a + f_b$$

Example 3.2.2.



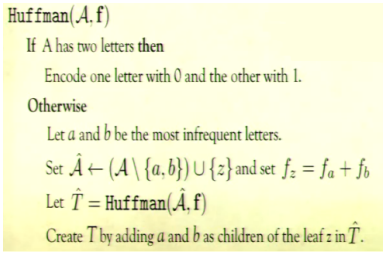
So now at this point we have:

1. Observation 1 → tells us that the two least frequent letters should be sibling leaves.
2. Observation 2 → tells us how to then recursively find the optimal shape of the tree.

We are now ready to introduce the main topic of this chapter:

3.3 Huffman's Algorithm

Algorithm 3.3.1.



See the example in the lecture to understand how this algorithm works!!!

Theorem 3.3.1. *The Huffman Coding Algorithm gives the minimum cost encoding.*

Proof. The base case is $|\mathcal{A}| = 2$. Here both letters have codewords of length 1, so no improvement can be made.

Now, our hypothesis is to assume that Huffman's Algorithm works if $|\mathcal{A}| = k$. Assume that $|\mathcal{A}| = k + 1$ and let a, b be the least frequent letters. Then, a, b are siblings in an optimal solution, and for any $\hat{T}$, $Cost(T) = Cost(\hat{T}) + f_a + f_b$. So the best choice for $\hat{T}$ is the optimal solution for $\hat{\mathcal{A}}$. But by our hypothesis, this is exactly the choice made recursively by the algorithm. □

3.3.1 Run-time

In this algorithm, there are $n - 2$ iterations. In each iteration it takes $O(n)$ to find the two least frequent letters and update the alphabet. Thus, the runtime is $O(n^2)$. Later in the course we will use a heap to implement this algorithm in $O(n \log n)$.

Acknowledgements

These notes have been transcribed from the lectures given by Prof. Adrian Vetta. Any figures are screenshots of the recorded lectures. This document is for personal use only and beware of typos!